Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №2

Технології паралельних обчислень

Тема: Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірила:
Стеценко І.В.

Київ 2024

ЗМІСТ

1 ЗАВДАННЯ

1. Реалізуйте стрічковий алгоритм множення матриць. Результат множення записуйте в об’єкт класу Result. 30 балів.
2. Реалізуйте алгоритм Фокса множення матриць. 30 балів.
3. Виконайте експерименти, варіюючи розмірність матриць, які перемножуються, для обох алгоритмів, та реєструючи час виконання алгоритму. Порівняйте результати дослідження ефективності обох алгоритмів. 20 балів.
4. Виконайте експерименти, варіюючи кількість потоків, що використовується для паралельного множення матриць, та реєструючи час виконання. Порівняйте результати дослідження ефективності обох алгоритмів. 20 балів.

2 ВИКОНАННЯ

2.1 Алгоритми

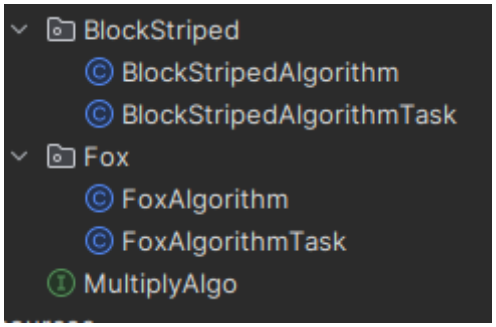


Рисунок 2.1.1 - Структура проєкту

Розглянемо структуру:

1) BlockStripedAlgorithm та FoxAlgorithm — класи паралельного множення матриць, у конструктори яких передається кількість потоків та дві матриці.
2) BlockStripedAlgorithmThread та FoxAlgorithmThread — класи, що надають можливість розділити множення матриць на потоки.
3) MultiAlgo — інтерфейс, що узагальнює використання алгоритмів.

Покажемо перемноження матриць на прикладі main-функції класу FoxAlgorithm, де виконується множення матриць за допомогою FoxAlgorithm та BlockStripedAlgorithm.

```
    public FoxAlgorithm(int threadsNum, Matrix2D first, Matrix2D second) { super(thr

    public static void main(String[] args) {
        var matrixFactory = new Matrix2DFactory();
        var rows = 3;
        var cols = 3;
        var minVal = 0;
        var maxVal = 10;
        var threadsNum = 3;

        var first : Matrix2D = matrixFactory.getRandom(rows, cols, minVal, maxVal);
        var second : Matrix2D = matrixFactory.getRandom(rows, cols, minVal, maxVal);

        var algorithm = new FoxAlgorithm(threadsNum, first, second);
        var striped = new BlockStripedAlgorithm(threadsNum, first, second);
        var result : Matrix2D = algorithm.solve();
        var stripedResult : Matrix2D = striped.solve();
        System.out.println("First:\t" + first);
        System.out.println("Second:\t" + second);
        System.out.println("Fox:\t" + result);
        System.out.println("Striped:\t" + stripedResult);
```

Рисунок 2.1.2 - Main-функція класу FoxAlgorithm

Покажемо результати виконання функції.

```
First:  {{1.0, 7.0, 0.0}, {6.0, 7.0, 7.0}, {1.0, 7.0, 3.0}}
Second: {{9.0, 7.0, 8.0}, {0.0, 6.0, 6.0}, {4.0, 3.0, 0.0}}
Fox:    {{9.0, 49.0, 50.0}, {82.0, 105.0, 90.0}, {21.0, 58.0, 50.0}}
Striped:    {{9.0, 49.0, 50.0}, {82.0, 105.0, 90.0}, {21.0, 58.0, 50.0}}
```

Рисунок 2.1.3 - Результат множення

2.2 Тестовий клас алгоритмів

Рисунок 2.2.1 - Структура проєкту

Розглянемо структуру:

1) MatrixTester — клас, що проводить тестування результатів часу множення матриць, варіюючи кількість потоків, розміри матриць, алгоритми та малюючи графік.
2) MatrixMulTester — клас, що проводить тестування правильності множення матриць.

Продемонструємо різницю роботи алгоритмів та часу виконання алгоритмів.

Під час виконання результати записуються у файл results.csv. Покажемо вміст цього файлу. У першій колонці присутня назва алгоритму, у другій — кількість потоків, у третій — розмір матриці, у четвертій — час в мілісекундах, у п’ятій — прискорення у порівнянні з одним потоком.

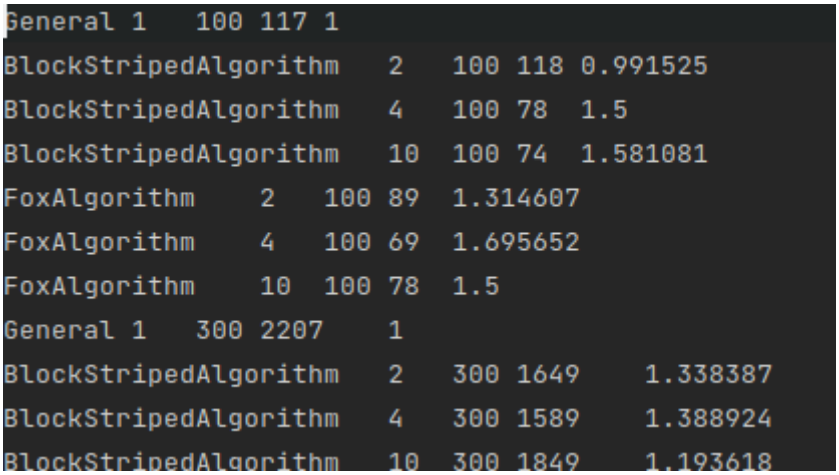
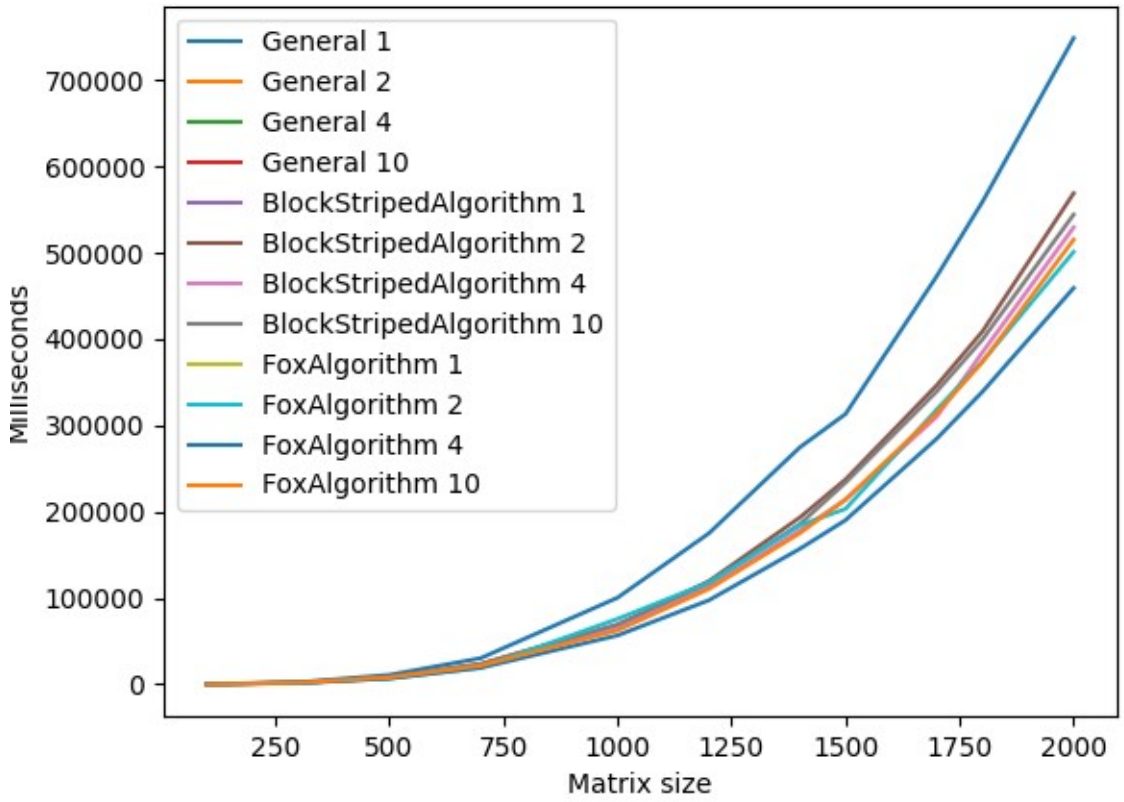


Рисунок 2.2.2 - Результати алгоритму

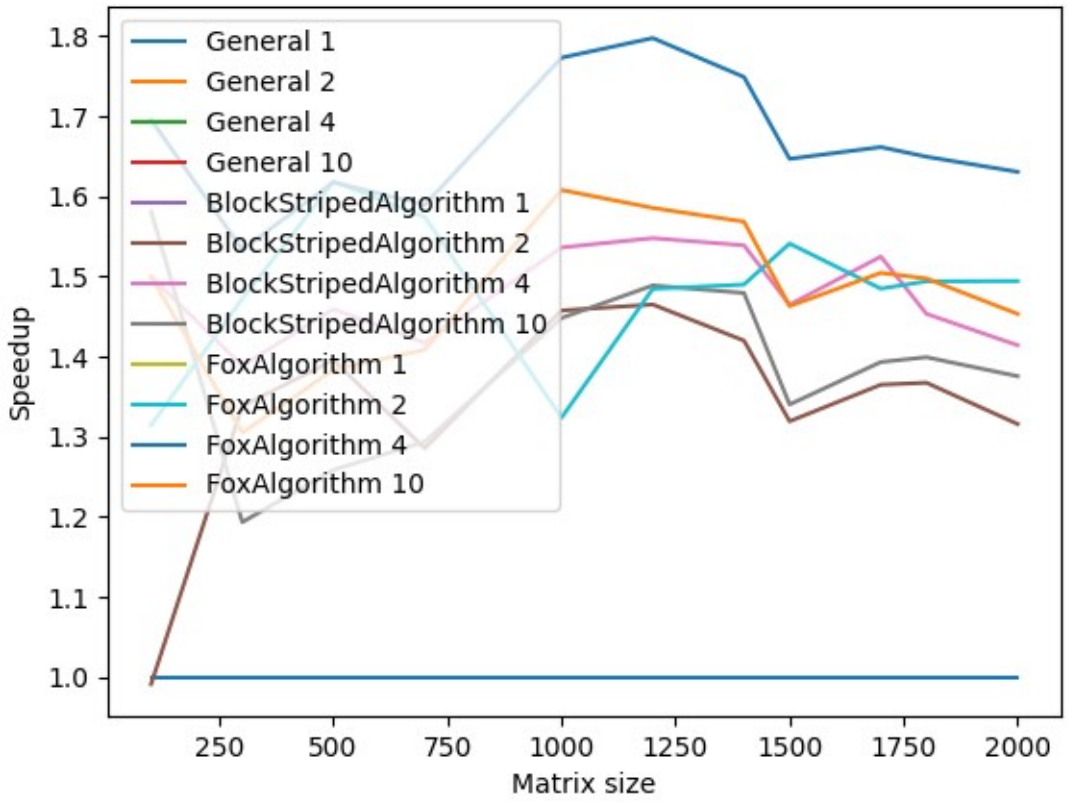
Не досить зрозуміло, тому у додачу до файлу будується графік та зберігається у файл “plot.png”.



У легенді позначено алгоритми та кількість потоків, що використовувалася.

Бачимо, що зі збільшенням розмірності матриць для усіх алгоритмів збільшується час їхнього виконання, а особливо це помітно для звичайного алгоритму, що працює на одному потоці.

Покажемо на прикладі прискорення, що зазнали алгоритми.



2.3 Математична бібліотека

Попередньо з першою лабораторною роботою дописав функціонал математичної бібліотеки, розділивши її на інтерфейси, перерахунки, матриці, вектори.

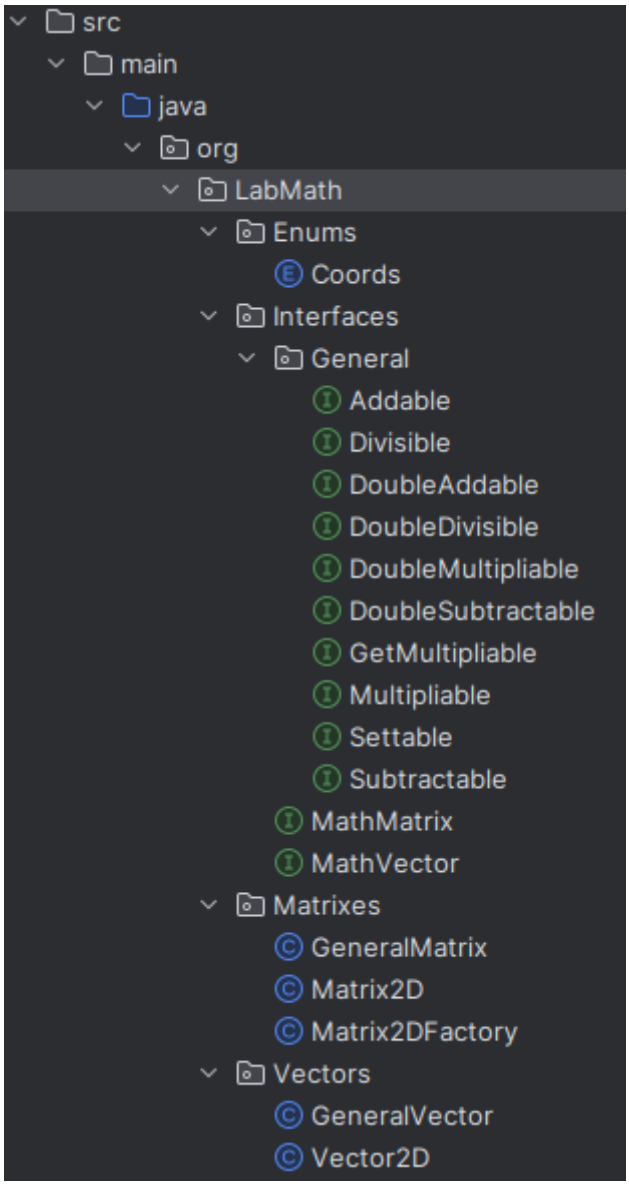


Рисунок 2.3.1 - Структура проєкту.

Розглянемо структуру:

1) Enums — пакет, що вміщує всі enum`и, що використовуються(Coords — enum координат);
2) Interfaces — пакет, що вміщує прості інтерфейси General, які описують певну просту властивість класу, що їх реалізує, та складні, як-от: MathMatrix, MathVector;
3) Mtrixes та Vectors — пакети, що включають в себе фабрики та класи матриць і векторів відповідно.

3 ВИСНОВОК

Під час лабораторної роботи опрацювали завдання з розробки паралельних алгоритмів множення матриць та дослідження їхньої ефективності. Результати записані до файлу, побудовано відповідні графіки. Було продемонстровано, що збільшення розмірності матриць суттєво вливає на час виконання алгоритму, а збільшення потоків зменшує час виконання операції.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-113 курсу

Панченка С. В

```
// ./Lab2/Lab2/src/main/java/org/LabMath/Enums/Coords.java

package org.LabMath.Enums;

public enum Coords {
    Y,
    X
}



// ./Lab2/Lab2/src/main/java/org/LabMath/Vectors/GeneralVector.java

package org.LabMath.Vectors;

import org.LabMath.Interfaces.MathVector;

import java.util.Arrays;

public class GeneralVector implements MathVector<GeneralVector> {
        private static final String ERROR_LENGTHS_NOT_EQUAL = "Lengths of
points arguments are not equal";
    private double[] arguments;

    public GeneralVector(int length) {
        setLength(length);
    }

    public GeneralVector(GeneralVector other) {
        setLength(other.getLength());
        set(other);
    }
```

```
public int getLength() {
    if(arguments == null) {
        return 0;
    }
    return arguments.length;
}

public void setLength(int length) {
    var currentLength = getLength();

    if(currentLength==length) return;

    var minLength = Math.min(currentLength, length);
    var args = new double[length];
    for(var i = 0; i < minLength; ++i) {
        args[i] = getAt(i);
    }

    arguments = args;
}

@Override
public void set(GeneralVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, other.getAt(i));
    }
}

@Override
public GeneralVector clone() {
    return new GeneralVector(this);
```

```
}

@Override
public String toString() {
    return Arrays.toString(arguments);
}

private void checkSizesEqual(GeneralVector other) {
    assert getLength() == other.getLength() : ERROR_LENGTHS_NOT_EQUAL;
}

@Override
public void add(GeneralVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) + other.getAt(i));
    }
}

@Override
public void add(double value) {
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) + value);
    }
}

@Override
public void sub(GeneralVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) - other.getAt(i));
    }
```

```
}

@Override
public void sub(double value) {
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) - value);
    }
}

@Override
public void mul(GeneralVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) * other.getAt(i));
    }
}

@Override
public void mul(double value) {
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) * value);
    }
}

@Override
public void div(GeneralVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) / other.getAt(i));
    }
}
```

```
@Override
public void div(double value) {
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) / value);
    }
}

@Override
public double getSize() {
    return Math.sqrt(getSizeSquared());
}

@Override
public double getSizeSquared() {
    double s = 0;
    for(var i = 0; i < getLength(); ++i) {
        s += Math.pow(getAt(i), 2);
    }
    return s;
}

@Override
public double getDotProduct(GeneralVector other) {
    checkSizesEqual(other);
    var prod = 0;
    for(var i = 0; i < getLength(); ++i) {
        prod += getAt(i) * other.getAt(i);
    }
    return prod;
}

@Override
```

```
public double getDistance(GeneralVector other) {
    checkSizesEqual(other);
    var dist = 0.0;
    for(var i = 0; i < getLength(); ++i) {
        dist += Math.pow(getAt(i) - other.getAt(i), 2);
    }
    dist = Math.sqrt(dist);
    return dist;
}

@Override
public GeneralVector getForwardVector() {
    var forwardVec = clone();
    var size = getSize();
    for(var i = 0; i < getLength(); ++i) {
        forwardVec.setAt(i, getAt(i) / size);
    }
    return forwardVec;
}

@Override
public double getAt(int index) {
    return arguments[index];
}

@Override
public void setAt(int index, double value) {
    arguments[index] = value;
}

@Override
public GeneralVector getOpposite() {
```

```
        var v = clone();
        v.toOpposite();
        return v;
    }

    @Override
    public void toOpposite() {
        for(var i = 0; i < getLength(); ++i) {
            setAt(i, -getAt(i));
        }
    }

}


// ./Lab2/Lab2/src/main/java/org/LabMath/Vectors/Vector2D.java

package org.LabMath.Vectors;

import org.LabMath.Enums.*;
import org.LabMath.Interfaces.MathVector;

public class Vector2D implements MathVector<Vector2D> {
    private final GeneralVector vec = new GeneralVector(2);

    public Vector2D() {}

    public Vector2D(double y, double x) {
        set(y, x);
    }

    public Vector2D(Vector2D other) {
```

```
    set(other);
}

public double getX() {
    return getAt(Coords.X.ordinal());
}

public double getY() {
    return getAt(Coords.Y.ordinal());
}

public void set(Vector2D other) {
    vec.set(other.vec);
}

public void set(double y, double x) {
    setX(x);
    setY(y);
}

public void setX(double value) {
    setAt(Coords.X.ordinal(), value);
}

public void setY(double value) {
    setAt(Coords.Y.ordinal(), value);
}

@Override
public Vector2D clone() {
    return new Vector2D(getY(), getX());
}
```

```
@Override
public String toString() {
    return vec.toString();
}

@Override
public void add(Vector2D other) {
    vec.add(other.vec);
}

@Override
public void add(double value) {
    vec.add(value);
}

@Override
public void sub(double value) {
    vec.sub(value);
}

@Override
public void sub(Vector2D other) {
    vec.sub(other.vec);
}

@Override
public void mul(Vector2D other) {
    vec.mul(other.vec);
}

@Override
```

```
public void mul(double value) {
    vec.mul(value);
}

@Override
public void div(Vector2D other) {
    vec.div(other.vec);
}

@Override
public void div(double value) {
    vec.div(value);
}

@Override
public double getSize() {
    return vec.getSize();
}

@Override
public double getSizeSquared() {
    return vec.getSizeSquared();
}

@Override
public double getDotProduct(Vector2D other) {
    return vec.getDotProduct(other.vec);
}

@Override
public double getDistance(Vector2D other) {
    return vec.getDistance(other.vec);
```

```
    }

    @Override
    public Vector2D getForwardVector() {
        var forwardVec = clone();
        forwardVec.vec.set(forwardVec.vec.getForwardVector());
        return forwardVec;
    }

    @Override
    public double getAt(int index) {
        return vec.getAt(index);
    }

    @Override
    public void setAt(int index, double value) {
        vec.setAt(index, value);
    }

    @Override
    public Vector2D getOpposite() {
        var v = clone();
        v.toOpposite();
        return v;
    }

    @Override
    public void toOpposite() {
        vec.toOpposite();
    }
}
```

```
// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/MathVector.java

package org.LabMath.Interfaces;

import org.LabMath.Interfaces.General.*;

public interface MathVector<T> extends Cloneable, Divisible<T>, Multipliable<T>,
Addable<T>, Subtractable<T>,
        DoubleDivisible, DoubleMultipliable, DoubleAddable, DoubleSubtractable {
    double getSize();
    double getSizeSquared();
    double getDotProduct(T other);
    double getDistance(T other);
    T getForwardVector();
    double getAt(int index);
    void setAt(int index, double value);
    T getOpposite();
    void toOpposite();
    void set(T other);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/MathMatrix.java

package org.LabMath.Interfaces;

import org.LabMath.Interfaces.General.*;

public interface MathMatrix<T> extends Cloneable, Addable<T>, Subtractable<T>,
Divisible<T>, Comparable<T>,
            GetMultipliable<T>, Settable<T>, DoubleSubtractable, DoubleMultipliable,
```

```
DoubleAddable, DoubleDivisible {
    int[] getDimensions();
    double getAt(int... indexes);
    void setAt(double value, int... indexes);
    int calcIndex(int... indexes);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/DoubleDivisible.java

package org.LabMath.Interfaces.General;

public interface DoubleDivisible {
    void div(double other);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/Divisible.java

package org.LabMath.Interfaces.General;

import jdk.jshell.spi.ExecutionControl;

public interface Divisible<T> {
    void div(T other) throws ExecutionControl.NotImplementedException;
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/Addable.java

package org.LabMath.Interfaces.General;
```

```
public interface Addable<T> {
    void add(T other);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/Subtractable.java

package org.LabMath.Interfaces.General;

public interface Subtractable<T> {
    void sub(T other);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/DoubleAddable.java

package org.LabMath.Interfaces.General;

public interface DoubleAddable {
    void add(double other);
}


//
./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/DoubleSubtractable.java

package org.LabMath.Interfaces.General;

public interface DoubleSubtractable {
    void sub(double other);
}
```

```
// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/Settable.java

package org.LabMath.Interfaces.General;

public interface Settable<T> {
    void set(T other);
}


//
./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/DoubleMultipliable.java

package org.LabMath.Interfaces.General;

public interface DoubleMultipliable {
    void mul(double other);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/Multipliable.java

package org.LabMath.Interfaces.General;

public interface Multipliable<T> {
    void mul(T other);
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Interfaces/General/GetMultipliable.java

package org.LabMath.Interfaces.General;
```

```
import jdk.jshell.spi.ExecutionControl;

public interface GetMultipliable<T> {
  T getMul(T other) throws ExecutionControl.NotImplementedException;
}


// ./Lab2/Lab2/src/main/java/org/LabMath/Matrixes/Matrix2DFactory.java

package org.LabMath.Matrixes;

public class Matrix2DFactory {

   public Matrix2DFactory() {}

   public static void main(String[] args) {
      var factory = new Matrix2DFactory();
      var minVal = 0;
      var maxVal = 10;
      var rows = 5;
      var cols = 6;
      var one = factory.getRandom(rows, cols, minVal, maxVal);
      var two = factory.getRandom(cols, rows, minVal, maxVal);
      var result = one.getMul(two);
      System.out.println(result);
   }

   public Matrix2D getRandom(int rows, int cols, int minVal, int maxVal) {
      var res = new Matrix2D(rows, cols);
      for(var i = 0; i < rows; ++i) {
         for(var j = 0; j < cols; ++j) {
```

```
                res.setAt(Math.random() * (maxVal - minVal) + minVal, i, j);
            }
        }
        return res;
    }
}
```

```
// ./Lab2/Lab2/src/main/java/org/LabMath/Matrixes/GeneralMatrix.java

package org.LabMath.Matrixes;

import org.LabMath.Interfaces.MathMatrix;
import org.LabMath.Vectors.GeneralVector;
import jdk.jshell.spi.ExecutionControl;

import java.util.Arrays;

public final class GeneralMatrix implements MathMatrix<GeneralMatrix> {
    private static final String ERROR_INDEXES = "Indexes are less than 0";
      private static final String ERROR_DIMENSIONS = "Matrix dimensions not equal";
     private static final String ERROR_DIMENSION_INDEXES = "Indexes length is not equal to amount of dimension";
    private final int[] dimensions;
    private final int total;
    private final GeneralVector mat;

    public GeneralMatrix(int... dimensions) {
        this.dimensions = dimensions.clone();
        var t = 1;
        for(var d : dimensions) t *= d;
```

```
    this.total = t;
    this.mat = new GeneralVector(this.total);
}

private String doDraw(int[] indexes, int dimension) {
    var res = new StringBuilder();
    res.append("{");
    for(var i = 0; i < this.dimensions[dimension]; ++i) {
        indexes[dimension] = i;
        if(dimension == this.dimensions.length - 1) {
            res.append(this.mat.getAt(calcIndex(indexes)));
        } else {
            res.append(doDraw(indexes, dimension + 1));
        }
        res.append(this.dimensions[dimension] - 1 == i ? "" : ", ");
    }
    res.append("}");
    return res.toString();
}

@Override
public String toString() {
    var indexes = new int[this.dimensions.length];
    return doDraw(indexes, 0);
}

private void checkDimensions(int[] dimensions) {
    if(!Arrays.equals(this.dimensions, dimensions)) {
        throw new IllegalArgumentException(ERROR_DIMENSIONS);
    }
}
```

```
private void checkIndexes(int[] indexes) {
    if(!Arrays.stream(indexes).allMatch(e -> e >= 0)) {
        throw new IllegalArgumentException(ERROR_INDEXES);
    }
}

@Override
public void add(GeneralMatrix other) {
    checkDimensions(other.dimensions);
    for(var i = 0; i < total; ++i) {
        this.mat.setAt(i, this.mat.getAt(i) + other.mat.getAt(i));
    }
}

@Override
public void add(double value) {
    for(var i = 0; i < this.total; ++i) {
        this.mat.setAt(i, this.mat.getAt(i) + value);
    }
}

@Override
public void div(double value) {
    for(var i = 0; i < this.total; ++i) {
        this.mat.setAt(i, this.mat.getAt(i) / value);
    }
}

@Override
public void mul(double value) {
    for(var i = 0; i < this.total; ++i) {
        this.mat.setAt(i, this.mat.getAt(i) * value);
```

```
        }
    }

    @Override
    public void sub(double value) {
        for(var i = 0; i < this.total; ++i) {
            this.mat.setAt(i, this.mat.getAt(i) - value);
        }
    }

    @Override
                public GeneralMatrix getMul(GeneralMatrix other) throws
ExecutionControl.NotImplementedException {
        throw new ExecutionControl.NotImplementedException("");
    }

    @Override
    public void set(GeneralMatrix other) {
        checkDimensions(other.dimensions);
        for(var i = 0; i < this.total; ++i) {
            this.mat.setAt(i, this.mat.getAt(i));
        }
    }

    @Override
    public void sub(GeneralMatrix other) {
        checkDimensions(other.dimensions);
        for(var i = 0; i < total; ++i) {
            this.mat.setAt(i, other.mat.getAt(i) - other.mat.getAt(i));
        }
    }
```

```
    @Override
    public int[] getDimensions() {
        return dimensions.clone();
    }

    @Override
    public double getAt(int... indexes) {
        checkIndexes(indexes);
        return this.mat.getAt(this.calcIndex(indexes));
    }

    @Override
    public void setAt(double value, int... indexes) {
        checkIndexes(indexes);
        var index = this.calcIndex(indexes);
        this.mat.setAt(index, value);
    }

    @Override
                        public void div(GeneralMatrix other) throws
ExecutionControl.NotImplementedException {
        throw new ExecutionControl.NotImplementedException("");
    }

    @Override
    public int calcIndex(int... indexes) {
        if(indexes.length != dimensions.length) {
            throw new IllegalArgumentException(ERROR_DIMENSION_INDEXES);
        }
        var index = 0;
        var mult = 1;
        for(var i : dimensions) mult *= i;
```

```
        for(var i = 0; i < indexes.length; ++i) {
            mult /= dimensions[i];
            index += indexes[i] * mult;
        }
        return index;
    }

    @Override
    public int compareTo(GeneralMatrix o) {
        for(var i = 0; i < total; ++i) {
            if((Math.abs(this.mat.getAt(i) - o.mat.getAt(i)) > 0.000000001)) {
                return 1;
            }
        }
        return 0;
    }
}
```

```
// ./Lab2/Lab2/src/main/java/org/LabMath/Matrixes/Matrix2D.java

package org.LabMath.Matrixes;

import org.LabMath.Interfaces.MathMatrix;
import jdk.jshell.spi.ExecutionControl;

public class Matrix2D implements MathMatrix<Matrix2D> {
    private static final String ERROR_MULTIPLICATION = "Rows and columns are
not equal";
    private static final String ERROR_INDEXES = "Indexes are less than 0";
    private final int rows;
    private final int cols;
```

```
    private final GeneralMatrix mat;

    public static void main(String[] args) {}

    @Override
    public String toString() {
        return mat.toString();
    }

    public Matrix2D(int rows, int cols) {
        mat = new GeneralMatrix(rows, cols);
        this.rows = rows;
        this.cols = cols;
    }

    public int getRows() {
        return this.rows;
    }

    public int getCols() {
        return this.cols;
    }

    @Override
    public void add(Matrix2D other) {
        this.mat.add(other.mat);
    }

    @Override
                        public void div(Matrix2D other) throws
ExecutionControl.NotImplementedException {
        throw new ExecutionControl.NotImplementedException("");
```

```
}

@Override
public void add(double value) {
   this.mat.add(value);
}

@Override
public void div(double value) {
   this.mat.div(value);
}

@Override
public void mul(double value) {
   this.mat.mul(value);
}

@Override
public void sub(double value) {
   this.mat.sub(value);
}

@Override
public Matrix2D getMul(Matrix2D other) {
   var cols = getCols();

   assert cols == other.getRows() : ERROR_MULTIPLICATION;

   var result = new Matrix2D(rows, cols);

   for(var i = 0; i < getRows(); ++i) {
      for(var j = 0; j < other.getCols(); ++j) {
```

```
            var value = 0.0;
            for(var k = 0; k < cols; ++k) {
                value += this.mat.getAt(i, k) * other.mat.getAt(k, j);
            }
            result.setAt(value, i, j);
        }
    }

    return result;
}

@Override
public void set(Matrix2D other) {
    this.mat.set(other.mat);
}

@Override
public void sub(Matrix2D other) {
    this.mat.sub(other.mat);
}

@Override
public int[] getDimensions() {
    return this.mat.getDimensions();
}

@Override
public double getAt(int... indexes) {
    if(indexes.length != 2) {
        throw new IllegalArgumentException(ERROR_INDEXES);
    }
    return this.mat.getAt(indexes);
```

```
    }

    @Override
    public void setAt(double value, int... indexes) {
        if(indexes.length != 2) {
            throw new IllegalArgumentException(ERROR_INDEXES);
        }
        this.mat.setAt(value, indexes);
    }

    @Override
    public int calcIndex(int... indexes) {
        return this.mat.calcIndex(indexes);
    }

    @Override
    public int compareTo(Matrix2D o) {
        return this.mat.compareTo(o.mat);
    }
}


// ./Lab2/Lab2/src/main/java/org/MultiplicationAlgorithms/MultiplyAlgo.java

package org.MultiplicationAlgorithms;

import org.LabMath.Matrixes.Matrix2D;

public interface MultiplyAlgo {
        Matrix2D multiply(int threadsNum, Matrix2D first, Matrix2D second);
}
```

```
//
./Lab2/Lab2/src/main/java/org/MultiplicationAlgorithms/Fox/FoxAlgorithmTask.jav
a

package org.MultiplicationAlgorithms.Fox;



import org.LabMath.Matrixes.Matrix2D;

import java.util.concurrent.Callable;

public class FoxAlgorithmTask implements Callable<Matrix2D> {
        private final Matrix2D first;
        private final Matrix2D second;
        private final Matrix2D result;

        public FoxAlgorithmTask(Matrix2D first, Matrix2D second, Matrix2D result){
                this.first = first;
                this.second = second;
                this.result = result;
        }

        @Override
        public Matrix2D call(){
                result.add(first.getMul(second));
                return result;
        }
}

// ./Lab2/Lab2/src/main/java/org/MultiplicationAlgorithms/Fox/FoxAlgorithm.java
```

```
package org.MultiplicationAlgorithms.Fox;

import org.LabMath.Matrixes.Matrix2D;
import org.MultiplicationAlgorithms.MultiplyAlgo;

import java.util.ArrayList;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FoxAlgorithm implements MultiplyAlgo {
    public FoxAlgorithm() {}

    @Override
    public Matrix2D multiply(int countThread, Matrix2D first, Matrix2D second) {
        var complexSize = (int) Math.sqrt(countThread - 1) + 1;
        var complexFirst = MatrixToComplexMatrix(first, complexSize);
        var complexSecond = MatrixToComplexMatrix(second, complexSize);

        var innerSize = complexFirst[0][0].getCols();
        var complex = new Matrix2D[complexSize][complexSize];
        for(var i = 0; i < complexSize; ++i) {
            for(var j = 0; j < complexSize; ++j) {
                complex[i][j] = new Matrix2D(innerSize, innerSize);
            }
        }

        var executor = Executors.newFixedThreadPool(countThread);
        for(var k = 0; k < complexSize; ++k) {
            var futures = new ArrayList<Future<Matrix2D>>();
            for(var i = 0; i < complexSize; ++i) {
                for(var j = 0; j < complexSize; ++j) {
```

```
                        var index = (i + k) % complexSize;
                        var task = new FoxAlgorithmTask(
                                complexFirst[i][index],
                                complexSecond[index][j],
                                complex[i][j]);
                        futures.add(executor.submit(task));
                    }
                }

                for(var i = 0; i < complexSize; ++i) {
                    for(var j = 0; j < complexSize; ++j) {
                        try {
                            complex[i][j] = futures.get(i * complexSize +
j).get();
                        } catch (Exception ignored) {}
                    }
                }
            }
            executor.shutdown();
            return ComplexMatrixToMatrix(complex, first.getRows(),
second.getCols());
        }

        private Matrix2D[][] MatrixToComplexMatrix(Matrix2D matrix, int size){
            var complexMatrix = new Matrix2D[size][size];
            var rows = matrix.getRows();
            var cols = matrix.getCols();
            var inner = ((cols - 1) / size) + 1;
            for(var i = 0; i < size; ++i) {
                for(var j = 0; j < size; ++j) {
                    complexMatrix[i][j] = new Matrix2D(inner, inner);
                    var local = complexMatrix[i][j];
```

```
                    for(var k = 0; k < inner; ++k) {
                        for (var l = 0; l < inner; ++l) {
                            var rowIndex = i * inner + k;
                            var colIndex = j * inner + l;
                            if(rowIndex >= rows || colIndex >= cols){
                                local.setAt(0, k, l);
                                continue;
                            }
                            var element = matrix.getAt(rowIndex, colIndex);
                            local.setAt(element, k, l);
                        }
                    }
                }
            }
            return complexMatrix;
        }

        private Matrix2D ComplexMatrixToMatrix(Matrix2D[][] complexMatrix, int rows, int cols) {
            var matrix = new Matrix2D(rows, cols);
            for(var i = 0; i < complexMatrix.length; ++i) {
                for(var j = 0; j < complexMatrix[i].length; ++j) {
                    var local = complexMatrix[i][j];
                    var localRows = local.getRows();
                    var localCols = local.getCols();
                    for (var k = 0; k < localRows; ++k) {
                        for (var l = 0; l < localCols; ++l) {
                            var rowIndex = i * localRows + k;
                            var colIndex = j * localCols + l;
                            if(rowIndex >= rows || colIndex >= cols) continue;
                            var el = local.getAt(k, l);
                            matrix.setAt(el, rowIndex, colIndex);
```

```
                        }
                    }
                }
            }

            return matrix;
        }
}
```

```
//                  ./Lab2/Lab2/src/main/java/org/MultiplicationAlgorithms/BlockStriped/
BlockStripedAlgorithm.java

package org.MultiplicationAlgorithms.BlockStriped;

import org.LabMath.Matrixes.Matrix2D;
import org.MultiplicationAlgorithms.MultiplyAlgo;

import java.util.concurrent.Future;
import java.util.ArrayList;
import java.util.concurrent.*;

public class BlockStripedAlgorithm implements MultiplyAlgo {
    protected static final String ERROR_MULTIPLICATION = "Rows and
columns are not equal";
    protected static final String ERROR_NUM_OF_THREADS = "Number of
threads must be positive";

    public BlockStripedAlgorithm() {}

    @Override
    public Matrix2D multiply(int threadsNum, Matrix2D first, Matrix2D second) {
```

```
        if(threadsNum <= 0) {
            throw new IllegalArgumentException(ERROR_NUM_OF_THREADS);
        }

        var firstRows = first.getRows();
        var firstCols = first.getCols();
        var secondRows = second.getRows();
        var secondCols = second.getCols();

        if(firstCols != secondRows) {
            throw new IllegalArgumentException(ERROR_MULTIPLICATION);
        }

        var result = new Matrix2D(firstRows, secondCols);

        var executor = Executors.newFixedThreadPool(threadsNum);
        var callables = new ArrayList<BlockStripedAlgorithmTask>();
        var futures = new ArrayList<Future<Double>>();

        for(var i = 0; i < secondCols; ++i) {
            //System.out.println("Iteration: " + i);
            for(var row = 0; row < firstRows; ++row) {
                var col = row - i;
                col = col < 0 ? col + secondCols : col;
                //System.out.println("\tProc: " + row + "\tCol: " + col);
                var task = new BlockStripedAlgorithmTask(row, col, first, second);
                callables.add(task);
            }
            try {
```

```
                futures.addAll(executor.invokeAll(callables));
                callables.clear();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }

        }
        executor.shutdown();
        try {
            for(var i = 0; i < secondCols; ++i) {
                for(var row = 0; row < firstRows; ++row) {
                    var col = row - i;
                    col = col < 0 ? col + secondCols : col;
                    var future = futures.get(i * firstRows + row);
                    result.setAt(future.get(), row, col);
                }
            }
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }

        return result;
    }
}

//              ./Lab2/Lab2/src/main/java/org/MultiplicationAlgorithms/BlockStriped/
BlockStripedAlgorithmTask.java

package org.MultiplicationAlgorithms.BlockStriped;

import org.LabMath.Matrixes.Matrix2D;
```

```
import java.util.concurrent.Callable;

public class BlockStripedAlgorithmTask implements Callable<Double> {
	private final Matrix2D first;
	private final Matrix2D second;
	private final int row;
	private final int col;

	public BlockStripedAlgorithmTask(int row, int col, Matrix2D first, Matrix2D second) {
		this.first = first;
		this.second = second;
		this.row = row;
		this.col = col;
	}

	@Override
	public Double call(){
		var element = 0.0;
		for(var i = 0; i < first.getCols(); ++i) {
			element += first.getAt(row, i) * second.getAt(i, col);
		}
		return element;
	}

}


// ./Lab2/Lab2/src/test/java/test/MatrixMulTester.java

package test;
```

```
import org.LabMath.Matrixes.Matrix2DFactory;
import org.MultiplicationAlgorithms.BlockStriped.BlockStripedAlgorithm;
import org.MultiplicationAlgorithms.Fox.FoxAlgorithm;

public class MatrixMulTester {
	private static final int minVal = -10;
	private static final int maxVal = 10;

	public static void main(String[] args) {
		var matrixSizes = new int[]{10, 20};
		var matrixFactory = new Matrix2DFactory();
		for(var size : matrixSizes) {
			var first = matrixFactory.getRandom(size, size, minVal, maxVal);
			var second = matrixFactory.getRandom(size, size, minVal,
maxVal);
			var generalRes = first.getMul(second);
			var striped = new BlockStripedAlgorithm();
			var fox = new FoxAlgorithm();

			var foxRes = fox.multiply(4, first, second);
			System.out.println(generalRes);
			System.out.println(foxRes);
			var stripedRes = striped.multiply(4, first, second);
			System.out.println(stripedRes);
			System.out.println(generalRes.compareTo(foxRes) == 0);
			System.out.println(foxRes.compareTo(stripedRes) == 0);
			System.out.println(generalRes.compareTo(stripedRes) == 0);
		}
	}
}
```

```
// ./Lab2/Lab2/src/test/java/test/MatrixAlgoTester.java

package test;

import com.github.sh0nk.matplotlib4j.*;
import org.LabMath.Matrixes.Matrix2DFactory;
import org.MultiplicationAlgorithms.BlockStriped.BlockStripedAlgorithm;
import org.MultiplicationAlgorithms.Fox.FoxAlgorithm;
import org.MultiplicationAlgorithms.MultiplyAlgo;

import java.io.*;
import java.util.ArrayList;

public class MatrixAlgoTester {
    private static final int minVal = -10;
    private static final int maxVal = 10;
    private static final String FILE_NAME = "results.csv";
    private static final String DELIMETER = "\t";
    private static final String LEGEND_POSITION = "upper left";
    private static final String X_LABEL = "Matrix size";
    private static final String Y_LABEL = "Milliseconds";
    private static final String SPEEDUP_Y_LABEL = "Speedup";
    private static final String PLOT_FILE = "plot.png";
    private static final String SPEEDUP_PLOT_FILE = "speedup_plot.png";


    private static class AlgorithmResult {
        public final long milliseconds;
        public final int threadsNum;
        public final int size;
        public final String name;
        public final double speedup;
```

```
        public AlgorithmResult(long time, int threadsNum, int size, double speedup,
String name) {
            this.milliseconds = time;
            this.threadsNum = threadsNum;
            this.size = size;
            this.name = name;
            this.speedup = speedup;
        }

        @Override
        public String toString() {
                return String.format("%s\t%d\t%d\t%d\t%f", name, threadsNum, size,
milliseconds, speedup);
        }
    }

        public static void main(String[] args) throws PythonExecutionException,
IOException {
            var tester = new MatrixAlgoTester();
            var threadsNums = new int[] {2, 4, 10};
               var matrixSizes = new int[] {100, 300, 500, 700, 1000, 1200, 1400, 1500,
1700, 1800, 2000};
            tester.testAlgorithm(threadsNums, matrixSizes);
            tester.plotStatistic();
            tester.plotSpeedup();
    }

    public void plotStatistic() throws IOException, PythonExecutionException {
        Plot plt = Plot.create();
        var results = readStatistic();
        var algorithms = results.stream().map(r -> r.name).distinct().toList();
```

```
        var threadNums = results.stream().map(r -> r.threadsNum).distinct().toList();
        for(var a : algorithms) {
            var filtered = results.stream().filter(r -> r.name.equals(a)).toList();
            for(var threadNum : threadNums) {
                var filteredByThreadNum = filtered.stream().filter(r -> r.threadsNum ==
threadNum).toList();
                var x = filteredByThreadNum.stream().map(r -> r.size).toList();
                var y = filteredByThreadNum.stream().map(r -> r.milliseconds).toList();
                plt.plot().add(x, y).label(a + " " + threadNum);
            }
        }
        plt.legend().loc(LEGEND_POSITION);
        plt.xlabel(X_LABEL);
        plt.ylabel(Y_LABEL);
        plt.savefig(PLOT_FILE);
        plt.show();
    }

    private void plotSpeedup() throws IOException, PythonExecutionException {
        Plot plt = Plot.create();
        var results = readStatistic();
        var algorithms = results.stream().map(r -> r.name).distinct().toList();
        var threadNums = results.stream().map(r -> r.threadsNum).distinct().toList();
        for(var a : algorithms) {
            var filtered = results.stream().filter(r -> r.name.equals(a)).toList();
            for(var threadNum : threadNums) {
                var filteredByThreadNum = filtered.stream().filter(r -> r.threadsNum ==
threadNum).toList();
                var x = filteredByThreadNum.stream().map(r -> r.size).toList();
                var y = filteredByThreadNum.stream().map(r -> r.speedup).toList();
                plt.plot().add(x, y).label(a + " " + threadNum);
            }
```

```
        }
        plt.legend().loc(LEGEND_POSITION);
        plt.xlabel(X_LABEL);
        plt.ylabel(SPEEDUP_Y_LABEL);
        plt.savefig(SPEEDUP_PLOT_FILE);
        plt.show();
    }



    private ArrayList<AlgorithmResult> readStatistic() throws IOException {
        var line = "";
        BufferedReader br = new BufferedReader(new FileReader(FILE_NAME));
        var results = new ArrayList<AlgorithmResult>();
        while ((line = br.readLine()) != null) {
            String[] row = line.split(DELIMETER);
            results.add(
                new AlgorithmResult(
                    Long.parseLong(row[3]),
                    Integer.parseInt(row[1]),
                    Integer.parseInt(row[2]),
                    Double.parseDouble(row[4]),
                    row[0]
                )
            );
        }
        return results;
    }

    public void testAlgorithm(int[] threadNums, int[] matrixSizes) throws IOException
{
        var file = new File( FILE_NAME);
```

```
var results = new FileOutputStream(file);
var matrixFactory = new Matrix2DFactory();
for(var size : matrixSizes) {
    var first = matrixFactory.getRandom(size, size, minVal, maxVal);
    var second = matrixFactory.getRandom(size, size, minVal, maxVal);
    var threadTimeOne = 0L;
    {
        var startTime = System.currentTimeMillis();
        first.getMul(second);
        threadTimeOne = System.currentTimeMillis() - startTime;
        var result = new AlgorithmResult(
            threadTimeOne, 1, size,
            threadTimeOne / (double) threadTimeOne , "General").toString();
        System.out.println(result);
        results.write((result + "\n").getBytes());
    }
    for(var i = 0; i < 2; ++i) {
        for(var threadsNum : threadNums) {
            var algName = "";
            MultiplyAlgo algo;
            if(i == 0) {
                algName = BlockStripedAlgorithm.class.getSimpleName();
                algo = new BlockStripedAlgorithm();
            } else {
                algName = FoxAlgorithm.class.getSimpleName();
                algo = new FoxAlgorithm();
            }
            var startTime = System.currentTimeMillis();
            algo.multiply(threadsNum, first, second);
            var duration = System.currentTimeMillis() - startTime;
            var result = new AlgorithmResult(
                    duration, threadsNum, size,
```

```
                    threadTimeOne / (double) duration , algName).toString();
                System.out.println(result);
                results.write((result + "\n").getBytes());
            }
        }
    }
}
```